Preprint
UCRL-JC-134444

Detecting Data and Schema Changes in Scientific Documents

I. Adiwijaya, T. Critchlow and R. Musick

This article was submitted to
16th International Conference on Data Engineering
San Diego, CA
February 28-March 3, 2000

June 8, 1999

Detecting Data and Schema Changes in Scientific Documents

Igg Adiwijaya[1], Terence Critchlow[2], and Ron Musick[2]
Rutgers University - CIMIC[1]
and
Lawrence Livermore National Laboratory (LLNL)[2]

Abstract

Data stored in a data warehouse must be kept consistent and up-to-date with the underlying information sources. By providing the capability to identify, categorize and detect changes in these sources, only the modified data needs to be transfered and entered into the warehouse. Another alternative, periodically reloading from scratch, is obviously inefficient. When the schema of an information source changes, all components that interact with, or make use of, data originating from that source must be updated to conform to the new schema. In this paper, we present an approach to detecting data and schema changes in scientific documents. Scientific data is of particular interest because it is normally stored as semi-structured documents, and it incurs frequent schema updates. We address the change detection problem by detecting data and schema changes between two versions of the same semi-structured document. This paper presents a graph representation of semi-structured documents and their schema before describing our approach to detecting changes while parsing the document. It also discusses how analysis of a collection of schema changes obtained from comparing several individual can be used to detect complex schema changes.

1 Introduction

Change detection is an important task for many applications, in particular for data warehouses. A data warehouses integrates data from heterogeneous, autonomous data sources into a consistent, central repository. Since the warehouse needs to be kept consistent and up-to-date, changes to the underlying sources must be periodically extracted and propagated to the warehouse. While this could be done by regularly refreshing the entire warehouse, a much better alternative is to detect and propagate only the changes. This alternative requires less computer resources and can be performed in significantly less time - an important consideration when warehouse down-time is limited.

Before data can be loaded, the source schema must be obtained and incorporated into the components supporting the warehouse, such as the wrapper and mediator [2]. When the schema evolves, these components must be modified as well. Currently, reflecting source schema changes in the warehouse requires obtaining the updated schema from the source and manually modifying the warehouse components to conform to it. While a manual approach to detecting schema changes may be acceptable in certain environments, it is too cumbersome and inefficient in situations where schema changes are frequent, such as scientific environment. An approach to automatically detecting schema changes, and semi-automatically modifying the warehouse components accordingly, is needed. The more this can be automated, the more useful it will be when adding new sources to the warehouse as well. However, since new schemata are usually provided in a free-form document, there are significant technical challenges to overcome if we are to reach any level of semi-automatic.

In this paper, we present our approach to detecting both data and schema changes in scientific data sources. In this environment, data is usually provided as semi-structured documents adhering to a well-defined, but representationally complex, schema. Therefore, we consider the problem of detecting change only within the context of semi-structured documents. To detect these changes, we compare two versions of the same document. The comparison utilizes any characteristics, rules and relationships existing in the documents, as described by the schema. We use a graph representation to describe the schema, and view documents as instances of this graph. Since a document contains data reflecting part of the schema (for example through tags), we use it as an indication of schema changes.

This paper makes the following contributions:

1. The definition of schema graphs as a new way to model semi-structured documents, and to allow more complete characterization of data and schema changes.
2. A set of algorithms that use the schema graph for the identification and characterization of schema and data changes.
3. The application of data analysis techniques to collections of document changes to identify complex schema modifications.
4. The observation that, in this approach, changes may be detected while the document is being parsed - resulting in a much more efficient algorithm.

The rest of this paper is organized as follows. We begin by briefly summarizing some of the traditional change detection algorithms. Then, in Section 3, we describe the characteristics of scientific documents, with an emphasis on genomics databases, and discuss how we formally represent them. Section 4, presents our approach to data change detection, including a description of the set possible data changes. Section 5 presents the set of possible schema changes and our approach to detecting them. Finally, we briefly present how this work relates to an ongoing data warehousing project at LLNL.

2 Related Work

There have been several papers which detect data changes by comparing two versions of the same document. Initial work in this area focussed on changes in unstructured documents. Myers [9, 14] detects changes between strings using the longest common subsequence (LCS) [7] algorithm, and consider only insertion and deletion operations. Wagner [12, 11] uses insertion, deletion and update operations to find the best sequence of operations that can transform one string into another.

Labio [8] proposes an approach to detecting data changes in legacy systems, which are unable to support triggers and log-files. In this approach, data is dumped into a flat-file with each line representing a unique record, including a key. In order to detect changes, two versions of the data file are sorted based on the keys. Then, the old and new versions are compared based on key value comparisons. For large data sources, this approach requires extensive computer resources unless a *windowing* algorithm is used. Windowing makes the assumption that data having the same key are stored in approximately the same location, relative to other entries, in both files, thus eliminating the necessity for sorting.

More recently, there have been several approaches to detecting data changes in semi-structured documents. These approaches differ primarily in how they view the underlying documents. Ball [5, 6] views semi-structured documents as containing sequence of sentences and "sentence-breaking" markups. A sentence is a non-recursive set of words and non-sentence-breaking markups. Sentence-breaking markups separate sentences from each other and from collections of sentences. In comparing two documents, LCS algorithm is used to determine the total number of matched words and markups within these sentences compared to the total length of the two sentences. With this approach, a sentence may need to be compared with all sentences in the other document. Even though changes to data can be detected, any possible changes to schema cannot be detected. This approach has been implemented in a system called *AT&T Internet Difference Engine* (AIDE) [5, 6].

Alternatively, Chawathe [3] and Zhang [10, 13, 15] view semi-structured documents as trees. Thus, the problem of change detection has been transformed to the problem of finding differences between trees. For any pair of leaf nodes, they either match or not, as determined by LCS. Two internal nodes strictly match if all their children match, and partially match if at least some children match. To detect data changes, the two documents are first converted into their corresponding trees, then the matching nodes are identified. Once this has been done, all of the sequences of operations (insert, delete, update, move) that convert the old tree to the new one are identified, usually resulting in several options. The sequence that best represents the transformation is the one with the lowest total cost, based on the cost assigned to each operation. This approach may be expensive since each node needs to be compared to all nodes in the other tree, and may match many of them, resulting in a large set of valid transformations.

Our approach to detecting change differs from these efforts in several ways. First, it does not require transforming one representation to another or finding the sequence of operations with minimum cost. Rather, we perform the tasks of node matching and comparison during the parsing of the documents. Second, none of the approaches previously mentioned detect changes to the underlying document schema, which our approach does. Finally, although we focus our discussion on scientific documents, our approach is applicable to semi-structured documents in general. The advantage of scientific documents is that they provide a more restrictive schema (i.e. fewer optional nodes) than most semi-structured documents. This restrictive schema allows us to identify a greater number of changes with less effort, as outlined in Section 5.3.

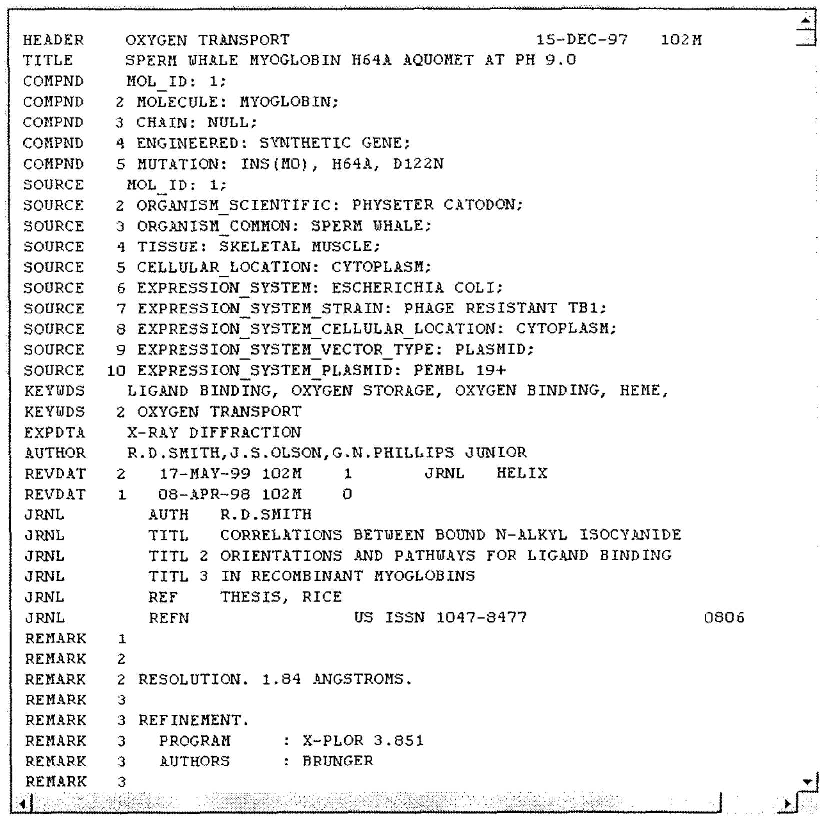
3 Semi-structured scientific documents

As described in Section 2, there are several ways one can view and model a semi-structured document. Our view is based on our experience with scientific documents in general, and genomic documents in particular. To provide a consistent and concrete framework for presenting our representation, we use a single example, the original Protein Data Bank (PDB) [1] data source shown in Figure 1. Before we discuss our model for scientific documents, we first present the characteristics and rules of the schema.

A schema for semi-structured scientific documents, s, consist of a set of data objects, O, and a set of constraints, C, between the data objects. A data object, o, is comprised of an identifier, $ident(o)$, and an optional value, $val(o)$. For example, in Figure 1, data object o_i with $ident(o_i) = COMPND$ has no value (i.e. $val(o_i) = null$) because there is no data associated with it, while object o_j, with $ident(o_j) = MOLECULE$, has a value of *MYGLOBIN*. In some cases, ordering among objects is significant, in others it is not. For example, the compound description must come after the title and before the source. However, it doesn't matter if the molecule name comes before the mutation information, or after it. Each schema object is contained in exactly one of the following sets:

1. *Mandatory objects.* These objects must exist in a document. For example, in a valid PDB document there must be a data object o_i with $ident(o_i) = HEADER$.
2. *Optional objects.* These objects may or may not exist in a valid document. For example, o_i where $ident(o_i) = REMARK4$ does not have to exist for a PDB document to be valid.

Objects participate in two types of relationships, *mandatory* and *optional*. In mandatory



```
HEADER    OXYGEN TRANSPORT                        15-DEC-97   102M
TITLE     SPERM WHALE MYOGLOBIN H64A AQUOMET AT PH 9.0
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: MYOGLOBIN;
COMPND   3 CHAIN: NULL;
COMPND   4 ENGINEERED: SYNTHETIC GENE;
COMPND   5 MUTATION: INS(M0), H64A, D122N
SOURCE    MOL_ID: 1;
SOURCE   2 ORGANISM_SCIENTIFIC: PHYSETER CATODON;
SOURCE   3 ORGANISM_COMMON: SPERM WHALE;
SOURCE   4 TISSUE: SKELETAL MUSCLE;
SOURCE   5 CELLULAR_LOCATION: CYTOPLASM;
SOURCE   6 EXPRESSION_SYSTEM: ESCHERICHIA COLI;
SOURCE   7 EXPRESSION_SYSTEM_STRAIN: PHAGE RESISTANT TB1;
SOURCE   8 EXPRESSION_SYSTEM_CELLULAR_LOCATION: CYTOPLASM;
SOURCE   9 EXPRESSION_SYSTEM_VECTOR_TYPE: PLASMID;
SOURCE  10 EXPRESSION_SYSTEM_PLASMID: PEMBL 19+
KEYWDS    LIGAND BINDING, OXYGEN STORAGE, OXYGEN BINDING, HEME,
KEYWDS   2 OXYGEN TRANSPORT
EXPDTA    X-RAY DIFFRACTION
AUTHOR    R.D.SMITH,J.S.OLSON,G.N.PHILLIPS JUNIOR
REVDAT   2   17-MAY-99 102M    1       JRNL   HELIX
REVDAT   1   08-APR-98 102M    0
JRNL        AUTH   R.D.SMITH
JRNL        TITL   CORRELATIONS BETWEEN BOUND N-ALKYL ISOCYANIDE
JRNL        TITL 2 ORIENTATIONS AND PATHWAYS FOR LIGAND BINDING
JRNL        TITL 3 IN RECOMBINANT MYOGLOBINS
JRNL        REF    THESIS, RICE
JRNL        REFN                  US ISSN 1047-8477                0806
REMARK   1
REMARK   2
REMARK   2 RESOLUTION. 1.84 ANGSTROMS.
REMARK   3
REMARK   3 REFINEMENT.
REMARK   3   PROGRAM     : X-PLOR 3.851
REMARK   3   AUTHORS     : BRUNGER
REMARK   3
```

Figure 1: Sample of a PDB document

relationships, if the *parent* exists in the instance of the schema being considered, the *child* must also exist. For *optional* relationships, this existence dependency does not hold. Obviously, mandatory objects may only participate in mandatory relationships. However, optional objects may participate in either mandatory or optional relationships, which allows conditions of the form: o_i does not need to exist an instance of the schema, however if it does, it must be followed by o_j This is extremely useful in scientific documents, where attributes may be optional, but if they exist they are well structured. For example, in a PDB document, a molecule name (*MOLECULE*) exists if and only if there is a corresponding compound description (*COMPND*).

Given these general characteristics of scientific documents, the rest of this section describes our representation of scientific documents and their schema. We view a schema as a directed acyclic graph (DAG), called a *schema graph*, and a document as an instance of it. Formally, we define a schema graph S as a DAG consisting of:

- Nodes, n.
  Nodes correspond to the schema objects previously defined. There are three different types of nodes: *regular, optional* and *stopping* nodes.
  - Regular nodes, n^r, must have an associated identifier. If an identifier is not explicitly specified in a document, a generated one will be supplied. Values are optional. For example, the identifier *COMPND* does not have a value while its children, such as *MOLECULE*, do. We use n_i^r and $ident(n_i^r)$ to denote a regular node i and its identifier respectively. A regular node is represented by a circle on the schema graph.

- Optional nodes, n^o, do not have identifiers or values. An optional node must have at least two children. For a given document, exactly one of the children nodes of an optional node must be selected. On a schema graph, an optional node is represented by a triangle.
- Stopping nodes, n^s, do not have identifiers or values. These nodes function as a no-op. Usually, one of the children of an optional node is a stopping node. A stopping node is represented by a circle with the letter S in it on the schema graph.

- Directed edges, e.
A directed edge connects two nodes, n_i and n_j, where the edge origins at n_i, the parent, and ends at n_j, the child, and $n_i, n_j \in \{n^r, n^o, n^s\}$. For any n_i whose immediate parent n_j is not an optional node, whenever n_j exists in a document, n_i must also exist. Such a relationship is denoted by $n_i \dashrightarrow n_j$. There are two types of directed edges; *ordered* and *unordered* edges.
  - Ordered directed-edges, e^o, indicate that the ordering of the children from left to right is significant and must be preserved. These edges are represented by a solid line on the schema graph.
  - Unordered directed-edges, e^u, indicate that the ordering of the children from left to right is not critical. These edges are represented by dashed lines on the schema graph.

Thus a schema graph is a tuple $\langle N, E \rangle$, where N is a set of nodes and E is a set of directed edges. Figure 2 depicts an example of a simple schema with the following properties and rules.

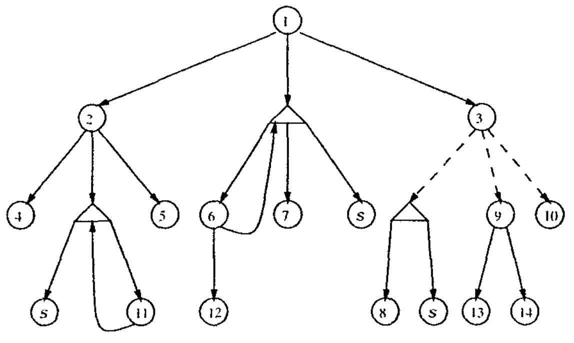


Figure 2: An example of a graph representation of a simple schema

$n_1 - n_{14}$ are regular nodes and $n_1, n_2, n_3, n_4, n_5, n_9, n_{10}, n_{13}, n_{14}$ are required. n_2 must appear before n_3 and, if present, (n_6, n_{12}) and n_7 must appear between n_2 and n_3. (n_9, n_{13}, n_{14}) and n_{10} may appear in any order after n_3. (n_6, n_{12}) and n_{11} can exist multiple times with different values.

We represent a semi-structured document by mapping it to an instance of the schema graph. The resulting *document graph* represents only a subset of the schema graph, since the optional rules and properties specified in the schema may not be reflected in the document. Figure 3 (a) shows a document graph conforming to the schema depicted by Figure 2. Because a document graph unrolls the loops in a schema graph, it is represented as a tree instead of a general graph structure. Figure 3 (b) depicts the tree schema, obtained by mapping the schema graph in Figure 2 to the document graph in Figure 3 (a).

As can be seen from Figure 3 (a), a document graph, denoted by t_d, consists of the following.

- Nodes, n. $n \in \{n^r, n^o, n^s\}$.
- Edges, e. $e \in \{e^o, e^u\}$.

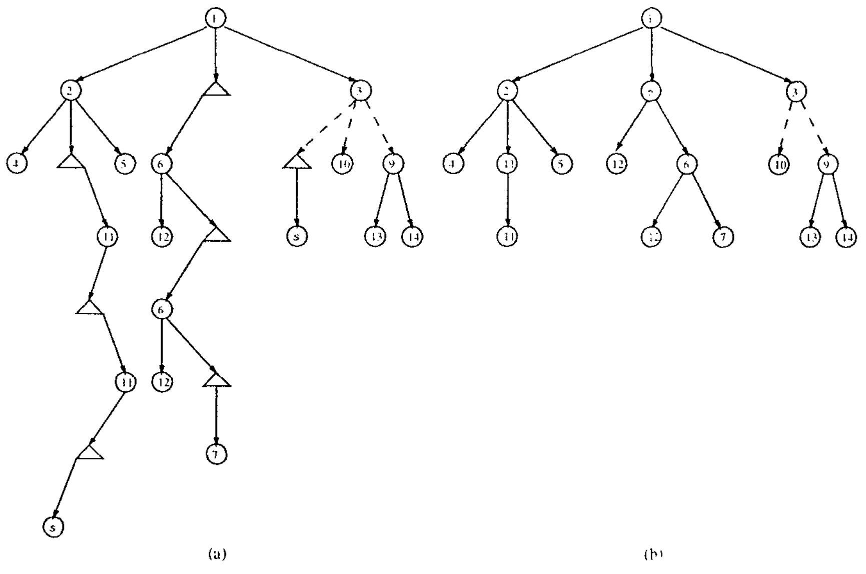


Figure 3: The subset graph and tree for a document

4 Changes to data

To detect data changes, we use the schema to guide the comparison between different versions of the document. This requires extending the current parser to read the original document and store it internally. This effectively combines the schema graph and the original document graph to produce a *value-added* schema graph. Then the new document is read using the value-added schema. with the data values being compared during the parsing. This allows us to detect and evaluate changes while the document is being loaded. Before we discuss detecting data changes in Section 4.2, we present our approach to producing the value-added schema and describe the types of changes we consider. Since a document graph is an instance of a schema graph, mapping the document graph to the schema graph is straightforward:

```
Parse the original document using the schema graph.
For every object within the document, mark the corresponding node in the schema graph.
    For every value in the document, copy it to the corresponding node.
    Extend optional nodes as necessary to handle loops.
```

To illustrate, suppose we would like to obtain the value added schema, $s^i_{d_j}$, where s^i is the schema given in Figure 2 and t_{d_j} is the document graph shown in Figure 3 (a). Figure 4 presents $s^i_{d_j}$ diagrammatically, with shaded circles and solid lines corresponding to the document's nodes and edges.

4.1 Types of changes to data

Next, we briefly describe the types of data changes we consider in our approach:

- **Update Value**: *upd*$(val(n_i))$ occurs when an update is made to the value of a node n_i. This change requires the same node to appear in both versions, and $n_i \in \{n^r\}$.
- **Insert node**: $ins(n_i)$ occurs when a node n_i does not exist in the original document but exists in the newer one.

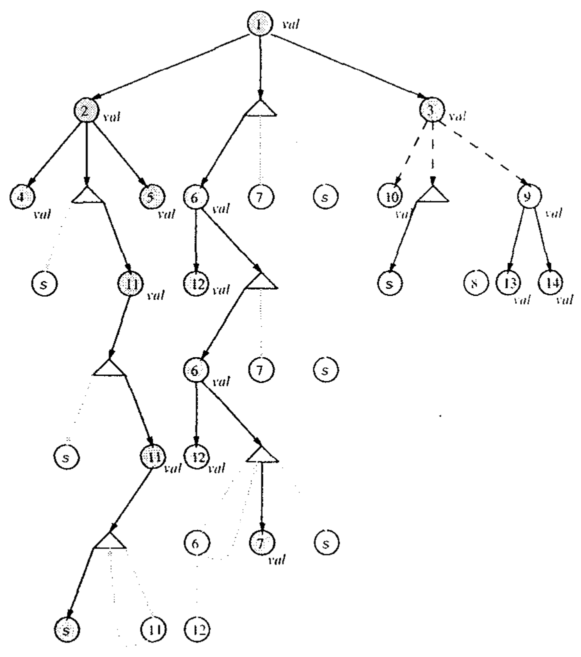


Figure 4: A schema having the knowledge of the content of a document

- **Delete node**: $del(n_i)$ occurs when an optional node n_i exists in the original document but not in the newer one. If n_i is a mandatory node, this change would indicate a schema change as discussed in the following section.
- **Reorder nodes**: $reorder(n_i)$ occurs when unordered If ordered nodes changed their position, this change would represent a schema change.
- **Move node**: $move(n_i)$ occurs when a node n_i is relocated upward or downward on the graph. This is only possible when the node is a child of an optional node.

4.2 Detecting changes to data

Before discussing our approach to detecting changes, we first clarify what we mean by two nodes matching. Two nodes match if:

1. the identifiers match, and
2. the values are *relatively* equal, where this implies they match under the LCS algorithm.

To detect data changes within a document, the value-added schema can be used to parse the document and return the updates. The following pseudo-code provides a high level overview of the algorithm we use to detect data changes. The remainder of this section describes, in detail, the various cases handled by this algorithm.

```
$S \longleftarrow s^i_{d^k_j}$ /* value added schema = schema + old d ocument */
$D \longleftarrow d^l_j$ */ new document */
While traversing $S$, parse $D$ using $S$
    for each $o_i \in D$
        find $o_j$, the corresponding node in $S$
        if $o_j$ is an ordered node then
    detect changes on ordered nodes $o_j$ and $o_i$
if $o_i$ is an unordered node **then**
    detect changes on unordered nodes $o_j$ and $o_i$
if $o_i$ is a non-repeatable, optional node **then**
    detect changes on non-repeatable, optional nodes $o_j$ and $o_i$
if $o_i$ is a repeatable, optional node **then**
    detect changes on repeatable, optional nodes $o_j$ and $o_i$
    the corresponding subtree sub-rooted at $o_j$
```

Ordered nodes
Detecting changes in ordered nodes is straightforward since they must occur in a specified sequence. Let d_j^k denote the kth version of document j. Given s^i, d_j^k, d_j^l $(k < l)$, to detect changes to ordered nodes in d_j^l:

Traverse $s^i_{d_j^k}$ while parsing d_j^l

For every ordered node $n_v \in s^i_{d_j^k}$ and its corresponding node $n_w \in d_j^l$

If val(n_v) does not match val(n_w), then return $upd(val(n_w))$

Unordered nodes
Given s^i, d_j^k, d_j^l $(k < l)$, to detect changes to unordered nodes in d_j^l:

Traverse $s^i_{d_j^k}$ while parsing d_j^l.

For each visited, mandatory unordered node n_v on $s^i_{d_j^k}$, fetch the corresponding object, n_w, in d_j^l.

Fetch n_{wc}, the children of n_w
Compare each of n_v's children with n_{wc} using only the identifier to find the corresponding node.
Record the order of the node and the object.
Compare the values of the matching node.
If the orders are different, return $reorder(n_w)$
If the values of matching nodes are different, add an $upd(n_w)$.

Non-repeatable, optional nodes
Given a schema graph s^i, a non-repeatable, optional node, n_k, is a node such that n_k's ancestor $n_l \in n^o$ (i.e. it is optional) and $\forall n_m \mid n_m$ is a descendent of n_l and $\neg\exists$ an edge from n_m to n_l (i.e. it is not in a loop). Given $s_i, d_j^k, d_j^l (k < l)$, to detect changes to non-repeatable, optional nodes:

Traverse $s^i_{d_j^k}$ and parse d_j^l.

For each visited, non-repeatable, optional node n_v

Identify the corresponding optional node, n_w in d_j^l.
If $ident(n_v) = ident(n_w)$ and $val(n_v) \neq val(n_w)$ then return $upd(n_w)$
If $ident(n_v) \neq ident(n_w)$ then return $del(n_v), ins(n_w)$
If $n_w = null$ then return $del(n_v)$.
If $n_v = null$ and $n_w \neq null$ then return $ins(n_w)$.

Repeatable, optional nodes
Given a schema graph s^i, a non-repeatable optional node, n_k, is a node such that n_k's ancestor $n_l \in n^o$ and $\exists n_m \mid n_m$ is a descendent of n_l and $\exists$ an edge from n_m to n_l. For example, Figure 5 (a) depicts an s^i showing an optional node having a repeatable child n_6. Figure 5 (b) depicts a portion of $s^i_{d^k_j}$ with repeated nodes, and Figure 5 (b) depicts the tree representation of d^l_j. To detect changes to repeatable nodes, we use two *buckets* as temporary storage during the change detection process: b^k_j is associated with d^k_j, b^l_j is associated with d^l_j.

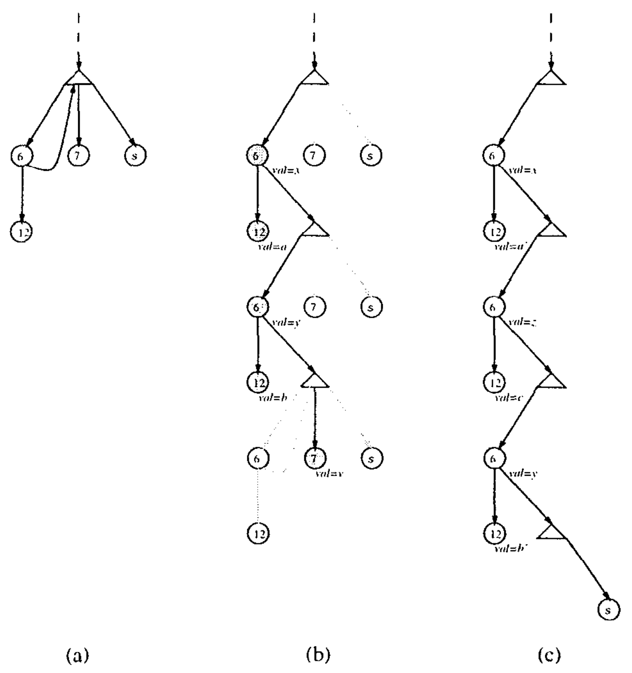


Figure 5: Detecting changes to repeatable nodes

```
Given $s^i, d^k_j, d^l_j$ $(k < l)$, to detect changes to repeatable, optional nodes:
  Traverse $s^i_{d^k_j}$ and parse $d^l_j$.
  For each node $n_m$ in $d^l_j$, compare each $n_m$ child, $n_{mc}$, with the child, $n_{sc}$, of the corresponding $n_s$ in $s^i_{d^k_j}$.
      If $n_{mc} = n_{sc}$, return $upd(n_{mc})$.
      If $n_{mc} \neq n_{sc}$
          compare $n_{mc}$ with $b^k_j$ sequentially.
          If a match is found, return $upd(n_{mc}), move(n_{mc})$ and remove $n_{mc}$'s matching from $b^k_j$. if a match is not found, append $n_{mc}$ to $b^l_j$.
             compare $n_{ms}$ with $b^l_j$ sequentially.
             If a match is found, return $upd(n_{ms}), move(n_{ms})$ and remove $n_{ms}$'s matching from $b^l_j$.
             if a match is not found, append $n_{ms}$ to $b^k_j$.
             return $del(n_v)$ for each $n_v \in b^k_j$ and return $ins(n_w)$ from each $n_w \in b^l_j$.
```

5 Changes to schema

To detect changes to schema, we need a document that conforms to the new schema and the current schema graph. Our approach is based on the observation that some, if not all, of the schema changes will be reflected in future documents. In particular, when the schema changes, some of the documents will fail to conform to the existing schema. As a result, changes that would normally be considered errors, such as an unrecognized identifier, are treated as schema modifications instead. In order to obtain the best reflection possible of the scope of schema changes, parsing should continue as much as possible after a schema change has been identified.

5.1 Types of schema changes

Before we describe our approach to detecting schema changes, we present the different categories schema changes we consider in our approach:

1. **Reordering a node**. For a given node, if the ordering of its children is significant, a reorder would occur if it is changed. For example, in future PDB documents, the ordering between *COMPND* and *SOURCE* may be switched. We use *reorderS*() to denote this operation.
2. **Inserting a new node**. We use *insS*() to denote this operation. An insert is considered to have taken place when an unrecognized identifier is found.
3. **Deleting a node**. For a regular node, a deletion is indicated by a document not containing a required child node. However, this may also indicate that it has been converted into an optional node. These options can only be differentiated by evaluating a set of documents. Similarly, the deletion of an optional node requires an analysis of multiple documents. Our approach to this analysis is briefly outlined in Section 5.3. We use *delS*() to denote a deletion of a node.
4. **Updating a node**. If an unrecognized identifier matches a previously existing child of the same parent, we consider that to be an update of the existing identifier. We use *updS*() to denote an update of a node on the schema.
5. **Adding/deleting a repeatable edge**. This schema change is only applicable to a regular node whose ancestor is an optional node. An addition of a repeatable edge can be detected by evaluating a single document where a node which previously never appeared multiple times now does The appearance of multiple nodes that were previously mutually exclusive may also indicate an addition of a repeatable edge. For example, given the schema example as depicted by Figure 2 the appearance of n_7, n_6, n_{12} in a new document would signal the insertion of a repeatable edge on n_7. Detecting the removal of a repeatable edge is more difficult task, and is described in section 5.3. We use *repeatS*() and *nonrepeatS*() to denote an addition and a deletion of a repeatable edge respectively.

5.2 Detecting changes to schema

For a node in the schema graph and an object in the document to *exactly* match, their identifier must be identical. Two nodes having different identifier partially match if more than some specified percentage of their children match, based on a sequential matching among the children in which order is not considered. The number of children that must match in order for the parents to match can be adjusted depending on the level of accuracy desired.

For the remainder of this section, we assume that the document we are parsing conforms to a newer version of the schema than the parser. We first present the general algorithm for detecting schema changes, then describe in detail the approach for each type of nodes. Section

5.3 discusses how we infer complex schema changes, taking into consideration the possibility of errors, by analyzing schema changes from a collection of incomplete document comparisons.

This algorithm traverses the schema graph identifying schema changes while parsing a document:

```
$S \longleftarrow$ root node of the schema graph
$D \longleftarrow$ the beginning or root object of the document
**while** $S = \emptyset$ **and** $D = \emptyset$ **do**
    $d \longleftarrow pop(D)$
    $s \longleftarrow pop(S)$
    **if** $s$ is a regular node **then**
      detect schema changes between $d$'s children and $s$'s children
      **for** any matching $d$'s children, $n_d$, with the coressponding children , $n_s$ of $s$ **do**
          $push(n_d, D)$ and $push(n_s, S)$
    **if** $s$ is an optional node **then**
      detect schema changes between $d$'s children $n_d$ and $s$'s children
      **if** $n_d$ matches a child-node, $n_s$, of $s$ **then**
          $push(n_d, D)$ and $push(n_s, S)$
          **if** $n_d$ is a repeatable node **then**
          $n_{dx} \leftarrow$ location in the document immediately following $n_d$
            $push(n_{dx}, D)$ and $push(n_s, S)$
```

S and D are list of nodes. *push*(y, L) and *pop*(L) have the standard FIFO queue semantics.

When detecting schema changes, this algorithm heavily uses the schema associated with the document. The more rigid the schema (i.e. the fewer optional nodes), the more changes we can detect. Thus while this approach will work for semi-structured documents in general, it is most useful when applied to documents that are well constrained - such as scientific documents. For example, in PDB documents, the beginning and ending of a major identifier, such as *SOURCE* is easily identified by its tag along the left column. Additionally, characters such as {:;.,} and "tab" can be used to identify the beginning/ending of a data object or grouping of data objects.

Children of a regular node
Given node n_k from schema graph s^i and the matching object n_l within a document d_m where both n_k and n_l are regular nodes, to detect schema changes to children of n_k:

> Traverse the sub-graph rooted at n_k creating a list, C_k, of n_k's chidren in order from left-to-right.
>
> Identify äll chilren of n_l, C_l, by parsing d_m starting from n_l until an identifier that matches one of the n_k's siblings is encountered.
>
> Sequentially compare C_k and C_l to detect changes.
>
> For ëvery $n_v \in C_k$ that strictly matches an identifier, $n_w \in C_l$
>
> > if order of n_k's children on s^i is significant, then $reorderS(n_w)$ is returned.
> >
> > For ëvery unmatched identifier in $n_w \in C_k$ and $n_v \in C_l$
> >
> > > If n_v partially matches n_w, $updS(n_w)$ is returned.
> > >
> > > For every unmatched n_v, $insS(n_v)$ is returned.
> > >
> > > For every unmatched n_w, $delS(n_w)$ is returned.

Children of an optional node
Given an optional node, n_k, on schema graph s^i, and the matching object, n_l, within a document d_m, to detect schema changes to children of n_k:

```
Traverse the sub-graph rooted at $n_k$ identifying all children of $n_k$, $C_k$.
Fetch the next identifier, $n_v$, immediately following $n_l$ in $d_m$.
    If $n_v$ macthes one of $n_k$'s sibling nodes, $n_l$'s children is a stopping node and no schema
    change is detected.
    If $n_v$ matches one of $n_k$'s children, $n_w$
        Fetch the next identifier from $d_m$. If it matches one of $n_k$'s children, $n_v$ is repeat-
        able.
        If $n_w$ is non-repeatable in $s^i$ and $n_v$ is a repeatable, then return $repeatS(n_v)$.
        If $n_v$ does not match any of $n_k$'s children, return $insS(n_v)$.
        If $n_w$ is a repeatable node that has been changed into a non-repeatable node, the
        approach described in Section 5.3 is used.
```

5.3 Inferring schema changes

A document conforming to a new version of a schema may not reflect all changes between the original schema and the new schema because the document is only an instance of the schema. Thus, all schema changes cannot be detected by evaluating a single document. In order to detect changes that span multiple documents, we envision applying data mining techniques to a large collection of comparison results. We believe that by using a large number of comparisons, we will be able to identify patterns reflecting the new schema.

The larger the number of documents considered, the greater our confidence in the resulting patterns. For example, assume that we have n documents that conform to schema s^{i+1}. By comparing these documents against the schema s^i, we create a set of schema changes for each document Ch where $Ch = \{ch_{d_1}, ch_{d_2}, ..., ch_{d_n}\}$. We can then use statistical analysis to answer the following problems.

1. *Determining when to delete a child of an optional node.*
   If n_l is the child of an optional node we are considering for deletion and if Ch indicates that data objects corresponding to n_l do not appear in any of the documents, we may deduce with a level of confidence proportional to n, that n_l has been deleted.
2. *Identifying unintended errors.*
   An error may be reflected by an $insS(), delS()$ or $upd()$ on a node. In Ch, an unintended schema change is likely to take place in only one or two documents. We could assert that schema changes occurring only in a very small percentage of documents are errors and ignore them. Because this could also ignore valid schema changes, we assume that the author makes only relatively minor errors.

We believe mining schema changes is a feasible approach to identifying complex modifications. However, human intervention may be required to resolve some conflicts. The main objective of this effort is to automatically detect as many schema changes as possible, so manual reconstruction of the entire new schema from scratch is no longer necessary.

6 Implementation within DataFoundry

We are currently implementing our approach within the context of the DataFoundry [4] project at Lawrence Livermore National Laboratory (LLNL). DataFoundry is a data warehouse that integrates scientific data from several distributed, autonomous, heterogeneous information sources. DataFoundry responds to LLNL scientists' need for a uniform and semantically consistent interface to a variety of data. Figure 6 provides a simplified view of the DataFoundry architecture. The wrapper extracts scientific documents from the underlying information sources, such as

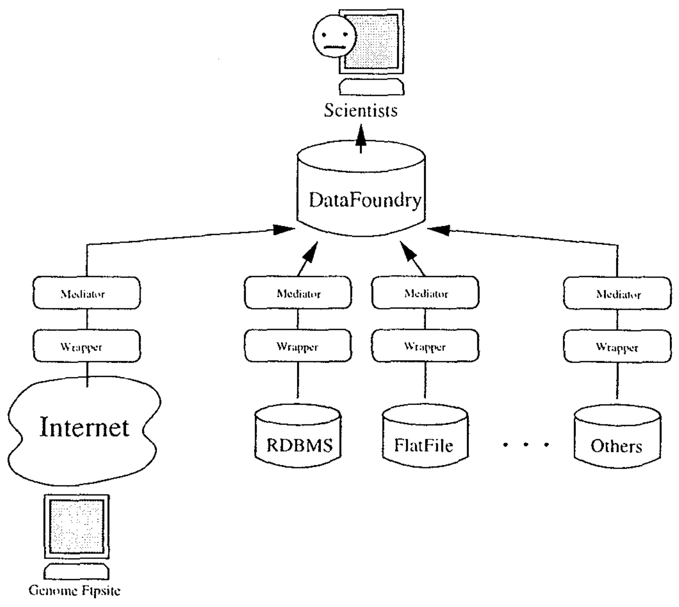


Figure 6: General Architecture for DataFoundry

PDB, parses them, and passes the data to the mediator. The mediator transforms it into the appropriate format, and propagates it to the warehouse. Because DataFoundry is focused on scientific data in general, and genomic data in particular, it faces the problems of detecting data and schema changes addressed in this paper. Typically, genomic information sources provide their schema as a free-formatted document, requiring manual identification of the locations and types of schema changes made to a new revision. Once these changes are identified, the associated wrapper component is manually updated to conform to the new schema. This has proven to be costly and time consuming. This work will extend DataFoundry by providing:

1. the wrapper with the capability to detect data and schema changes in a scientific document, and
2. a mechanism to semi-automatically define the new schema based on the older version, and to automatically incorporate the new schema into the wrapper.

To achieve the first objective, we are incorporating our change detection approach into the wrappers. We have developed a module that periodically detects and retrieves new documents from the information sources. These documents are then passed on to a set of C/C++ and Lex&Yacc programs to be parsed. We are currently extending these programs to utilize the value-added schema to detect data and schema changes. We anticipate completing our implementation and presenting detailed result in the final version of this paper.

To achieve the second objective, we propose developing a change management architecture for DataFoundry, shown in Figure 7. This architecture extends the current DataFoundry architecture by adding the *schema mining*, *schema storage* and *generator* components.

In this architecture, when a change is detected, the wrapper retrieves all newly created or modified documents from the information source. Each extracted document is compared

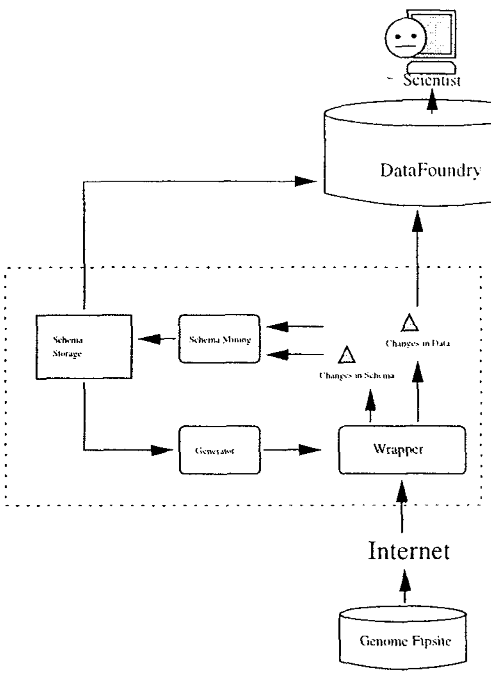


Figure 7: General Architecture for the change management system

with its older version, stored locally by the management system[1]. Data changes are extracted and propagated into DataFoundry while schema changes are stored. By comparing several documents, and accumulating the schema changes, we obtain a better view of the new schema. The schema mining component then analyzes the entire collection of schema changes and define a new schema based on the previous version. In doing so, human intervention may be required to resolve conflicts arising from incomplete or inconsistent data. The new schema is added to the schema storage, and is used by the generator to define a wrapper that conforms to it.

7 Conclusion

To update the content of a scientific data warehouse, changes to documents stored at the underlying information sources need to be extracted and incorporated into the warehouse. Moreover, if the schema of an underlying information source changes, the corresponding wrappers must be updated to conform to the changes. Thus, the ability to detect both data and schema changes is required in this environment.

In this paper we have highlighted the importance of detecting changes to both data and schema, and proposed a formal representation of semi-structured, scientific documents and their schema. We have addressed the problem of detecting data and schema changes by comparing the new document, with its implicit schema information, to the older version, represented as a value-added schema graph. We have presented the types of data and schema changes that may occur in scientific documents, and proposed detection algorithms accordingly. Our approach avoids performing extensive data matching between two versions of documents by performing

[1]In order to minimize storage requirement, only the newest version of each document is stored. Existing compression techniques can be used to further reduce the storage requirement

change detection during the parsing of the documents. and by using the schema to guide the process. In order to minimize manual intervention for detecting schema changes and modifying existing wrappers, we have proposed a general purpose architecture and necessary components for an automated change-management system. We are currently in the process of implementing this architecture within the context of the DataFoundry project at LLNL.

References

[1] Protein Data Bank. Protein Data Bank Contents Guide: Atomic Coordinate Entry Format. In *published at http://www.pdb.bnl.gov*, 1999.

[2] S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS Project: Integration of Heterogenous Information Sources. In *IPSJ Conference*, 1994.

[3] S. Chawathe, A. Rajaraman, H. Garcia-Molina. and J. Widom. Change Detection in Hierarchically Structured Information. In *Proceedings of the ACM SIGMOD Conference*, June 1996.

[4] T. Critchlow, M. Ganesh, and R. Musick. Meta-Data Based Mediator Generation. In *Proceedings of the Third IFCIS Conference on Cooperative Information Systems (CoopIS'98)*, 1998.

[5] F. Douglis and T. Ball. Tracking and Viewing Changes on the Web. In *1996 USENIX Technical Conference*, 1996.

[6] F. Douglis, T. Ball, and Y. Chen. WebGUIDE: Querying and Navigating Changes in Web Repositories. In *Fifth International World Wide Web Conference*, May 1996.

[7] D.S. Hirschberg. Algorithms for the longest common subsequence problem. In *Journal of the ACM*, pages 664–675, October 1977.

[8] W. J. Labio and H. Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *Proceedings of VLDB Conference*. September 1996.

[9] E. Myers. An O(ND) difference algorithm and its variations. In *Algorithmica*, volume 1, pages 251–266, 1986.

[10] D. Sasha and K. Zhang. Fast algorithms for unit cost editing distance between trees. In *Journal of Algorithms*, volume 11, 1990.

[11] R. Wagner. On the complexity of the extended string-to-string correction problem. In *Seventh ACM Symposium on the Theory of Computation*, 1975.

[12] R. Wagner and M. Fischer. The string-to-string correction problem. In *Journal of the Association of Computing Machinery*, volume 21, pages 168–173, January 1974.

[13] J. Wang, K. Zhang, K. Jeong, and D. Shasha. A System for Approximate Tree Matching. In *IEEE Transaction On Knowledge and Data Engineering*, volume 6, pages 559–570, August 1994.

[14] S. Wu, U. Manber, and E. Myers. An O(NP) sequence comparison algorithm. In *Information Processing Letters*, volume 35, pages 317–323. September 1990.

[15] K. Zhang, J. Wang, and D. Sasha. On the editing distance between undirected acyclic graphs. In *International Journal of Foundations of Computer Science*, 1995.

*This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.